第4章

車載LANのソフトウェア・テストを自動化する

――表計算ソフトを利用してLIN 2.0ドライバのテスト・プログラムを生成

名野 響

近年，自動車をはじめとする組み込みシステムの開発工程では，ソフトウェア開発の比重が高くなっている．中でも，コード量の増大や品質維持といった観点から，ソフトウェア・テスト工程におけるカバレッジや効率化が重要になっている．筆者らは車載ネットワークのソフトウェア・ドライバなどを開発しているが，テスト・プログラムの生成とテスト実行を自動化することで開発の効率を引き上げることを考えた．ここでは，筆者らが実際に開発したLIN 2.0通信ドライバを例にとり，ソフトウェア・テストの自動化手法について解説する．（編集部）

組み込みソフトウェアに対する機能と品質の要求は，増大の一途をたどっています．その一方で，納期短縮という要求もあります．こうした要求にこたえるため，開発の多くの工程で作業の改善（あるいは改革）が必要となっています．とくにソフトウェア・テストの工程は品質確保の最後のとりでです．要求仕様に対してもれなく効率的にテストを実施することが必要です（右掲のコラム「ソフトウェア・テスト自動化の現実」を参照）．

ここでは，LIN（local interconnect network）2.0のドライバ・ソフトウェアを例に，ソフトウェア・テストの効率化について考えます．現在，ほとんどの組み込みソフトウェアはなんらかのインターフェースによって他者（ハードウェアや上位アプリケーションなど）と通信しているはずです．今回取り上げるLIN 2.0は通信ソフトウェアですが，ほかのタイプのソフトウェアにも応用できると思います．

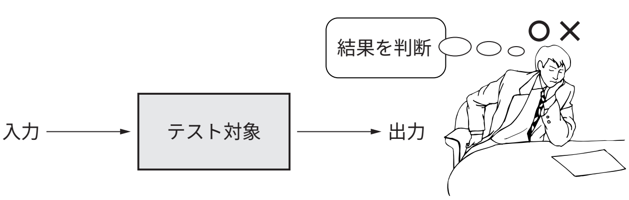


図1　テストの入力と出力

テストとは，テスト対象に与えた入力と得られた出力から結果を判断することである

1 自動化できる項目を見定める

一般に，テスト対象に対して何かしらの「入力」を与え，それによって得られた「出力」を判別することで，テストの合否を決定します（**図1**）．今回のテストでは，この「入力」と「出力」を次のように定めています．

- **入力：テスト実行プログラム＋テスト・パラメータ**
- **出力：テスト結果，テスト番号，エラー・コード**

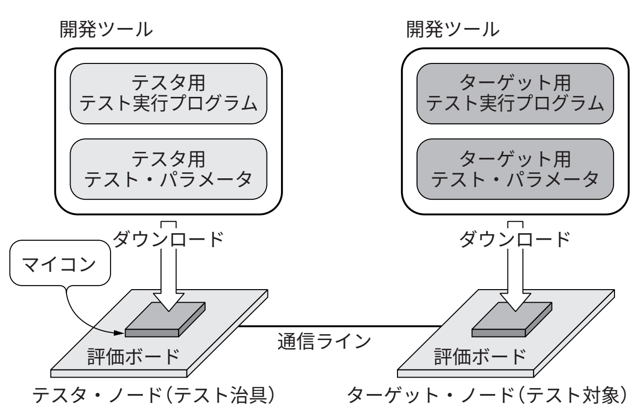


図2　テスト・システムの構成

テスト治具とテスト対象を通信ラインで接続し，テストを実施する

KeyWord　ソフトウェア・テスト，LIN 2.0，ドライバ，テスト・パラメータ，プログラム・ファイル，μPD78F0148H，78K0/KF1＋，μPD70F3239，V850ES/FJ2，PM＋，MULTI 2000，ビット・エラー

column ソフトウェア・テスト自動化の現実

「テストの自動化」．どこの開発現場でも品質向上とコスト削減の切り札としてこのことばを信奉しているはずです．しかし，実際のところ「言うは易く，行うは難し」といった状況ではないでしょうか．その原因は，大きく二つのカテゴリで考えられます（**表A-1**）．

1）物理的に自動化しにくいもの

2）担当者の意識の問題

さらに自動化する以前に，テスト項目をどのように（もれなく，効率良く）抽出すればよいかという問題があり，これが意外と悩みどころであると思います．本稿で紹介している事例は，要求仕様に対してもれなくテスト項目を抽出するという点に着眼しました．そして，自動化については市販のツールではなく手元にあるツール（今回はExcel）をくふうして使用したという点もポイントです．

さらに大きなシステムでは，「もれなく」の解決にまじめに取り組むと，天文学的な数のテスト項目を扱わなければならないという事態に陥ってしまい，これを「問題のないように」省く技術が必要になります．

表A-1 「自動化」を実現しにくい理由

カテゴリ	例
物理的に自動化しにくい	●動作をパターン化しづらいシステム ●入力を自動化しにくいシステム ●出力を読み取りにくい（人が読むしかない場合） ●良不良の判断が単純にできない入出力関係
担当者の意識の問題	●「自動化」と言われても，何を行えばよいのかわからない ●自動化する環境を作るくらいなら人手でやったほうが手っ取り早いと信じている ●自動化にはそれなりのツールを買うしかないと信じている

テストを実施するシステムの構成を**図2**に示します．テスト治具（ここではテスタ・ノードと呼ぶ）とテスト対象（ここではターゲット・ノードと呼ぶ）を通信ラインで接続します．そして，開発ツールを用いてテスト・パラメータとテスト実行プログラムをそれぞれのノードにダウンロード（入力）し，テストを実施します．

ここでテスト・パラメータとは，テストを実施するテスト治具やテスト対象（以降，これらを合わせて「テスト・システム」と呼ぶ）のふるまいを定義した値です（詳細は後述）．テスタ・ノードとターゲット・ノードでテストを実行するプログラムを共通にし，要求仕様に対応したふるまいごとのテスト項目をパラメータ化することで自動化を目ざしています．

テスト結果はツール上で変数の値を直接確認するか，またはログ・ファイルを参照して確認します．ほとんどの項目はログ・ファイルに記録でき，期待値との一致検証を行うことで完全に自動化できます．

今回のテストの流れを以下に示します（**図3**）．

1）**テスト項目の抽出** ―― テスト対象となる通信ドライバの仕様書からテスト項目を抽出し，テスト仕様書を作成する

2）**テスト・パラメータの作成** ―― 抽出した各テスト項目のテスト・パラメータを作成する

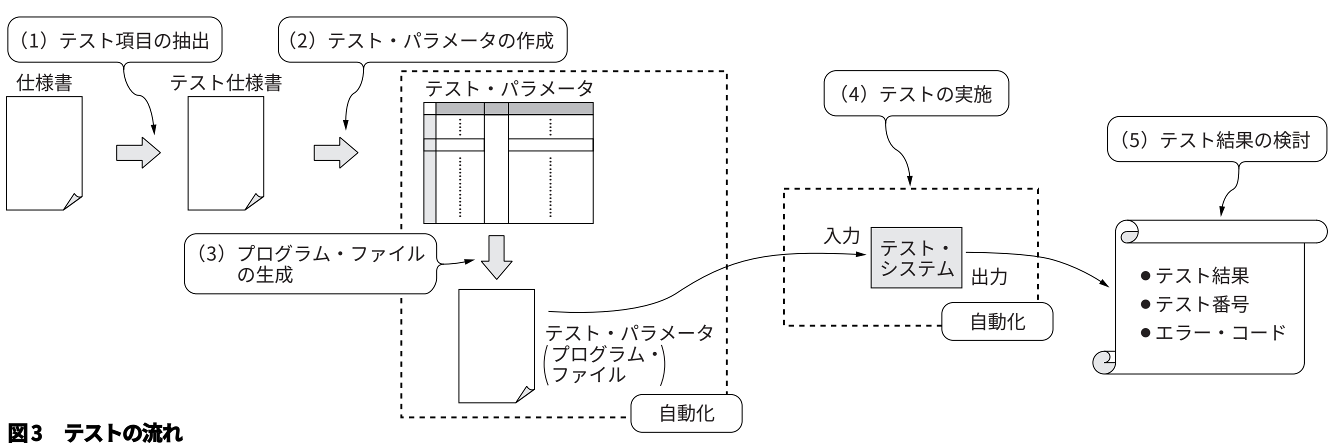


図3 テストの流れ

要求仕様からテスト項目をもれなく抽出する．その後，抽出したテスト項目からテストのふるまいを定義するテスト・パラメータを作成する．プログラム・ファイルの生成からテストの実行，テスト結果の出力までは自動で行う

3) **プログラム・ファイルの生成** ―― テスト・パラメータをテスト・システムに入力するプログラム・ファイルに変換する
4) **テストの実施** ―― プログラム・ファイルをテスト・システムに入力し，テストを実施する
5) **テスト結果の検討** ―― テスト・システムが出力した結果をもとに検討を行う

今回は上記の3)と4)を自動化します．

● 要求仕様とテスト仕様をもれなく一致させる

先に述べたように，テストとは「設計したソフトウェア（今回は通信ドライバ）が要求仕様を満たしているかどうかをもれなく確認すること」です．つまり，仕様書に記載されている各要求仕様を確認するためのテスト項目がかならず入っていなければなりません．

そこで，今回は要求仕様とテスト項目の関係をマトリックスを用いて整理しました（**図4**）．この作業を行うことで，各テスト項目がどの要求仕様を確認するためのものなのかをひと目で確認でき，テストもれの防止につながります．マトリックスの横軸にあたる要求仕様項目は，仕様書の記述を完全に網羅するように抽出したものです．プログラムを設計する際も，同じようにマトリックスを用いて要求仕様項目と実現モジュールの対応をとります．これによって要求仕様と設計，要求仕様とテスト仕様の関係を一つの手法で明示しつつ，もれ防止を実現しています．

図5に表計算ソフトウェア（Excel）を用いて作成したマトリックスの例を示します．要求仕様とテスト項目の対応関係はマトリックスに○印で表します．ある要求仕様に対して○印がまったくなければ，その仕様についてのテストがもれていることになります．

● 効率良く，信頼性の高いテスト・プログラムを作る

要求仕様から抽出したテスト項目は膨大な量になります．そのため，一つ一つのテスト項目に対応したテスト・パターン（プログラム）を作成していては時間がいくらあっても足りません（**図6**）．この問題を解決するため，今回は次のような手法をとっています．

- **テスト・プログラムの共通化** ―― テスタ・ノードとターゲット・ノードごとのテスト実行プログラムは，すべてのテスト項目について同じものを使用する．このプログラムは次に説明するテスト・パラメータを読み込んで動作する

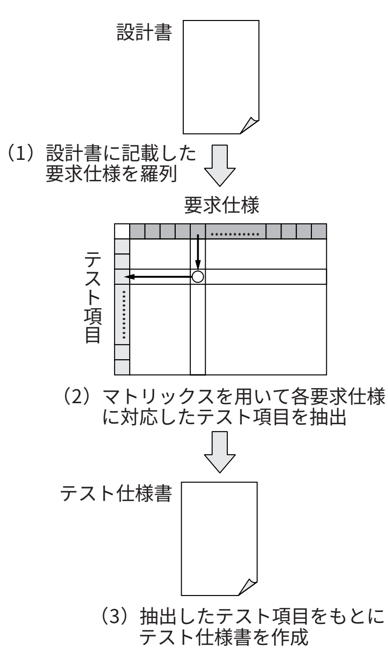
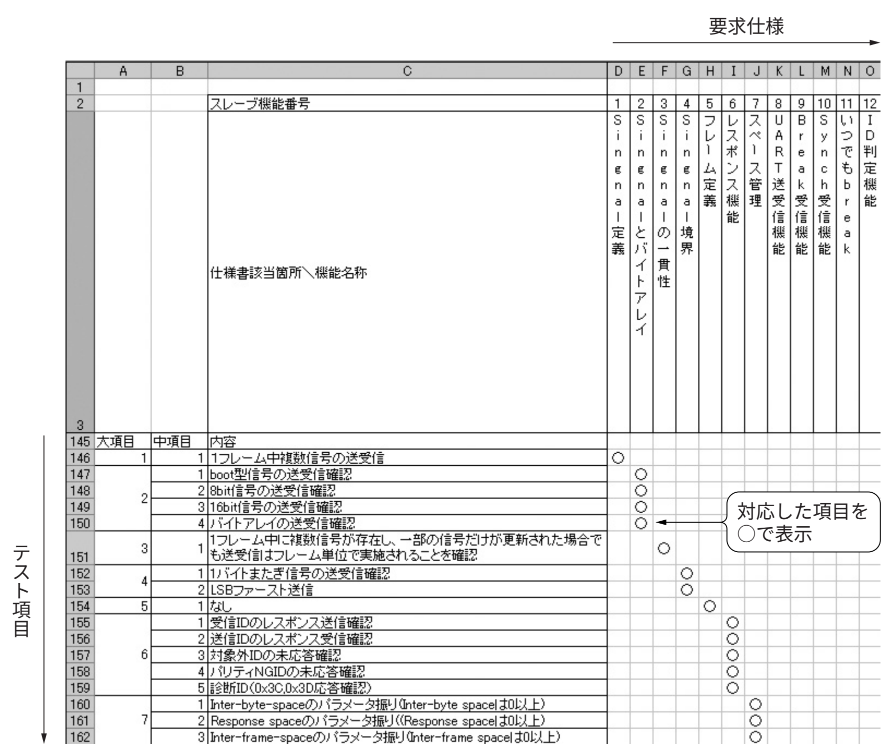


図4　要求仕様とテスト項目のマトリックス
要求仕様とテスト項目の対応関係をマトリックスで示す．こうすることでテスト項目のもれを防ぐことができる



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1															
2			スレーブ機能番号	1	2	3	4	5	6	7	8	9	10	11	12
3			仕様書該当箇所＼機能名称	Single定義	Singleとバイトアレイ	Singleの一貫性	Single境界	フレーム定義	レスポンス機能	スペース管理	UART送受信機能	Break受信機能	Synch受信機能	いつでもbreak	ID判定機能
145	大項目	中項目	内容												
146	1	1	1フレーム中複数信号の送受信	○											
147	2	1	boot型信号の送受信確認		○										
148		2	8bit信号の送受信確認		○										
149		3	16bit信号の送受信確認		○										
150		4	バイトアレイの送受信確認		○										
151	3	1	1フレーム中に複数信号が存在し、一部の信号だけが更新された場合でも送受信はフレーム単位で実施されることを確認			○									
152	4	1	1バイトまたぎ信号の送受信確認				○								
153		2	LSBファースト送信				○								
154	5	1	なし					○							
155	6	1	受信IDのレスポンス送信確認						○						
156		2	送信IDのレスポンス受信確認						○						
157		3	対象外IDの未応答確認						○						
158		4	パリティNGIDの未応答確認						○						
159		5	診断ID(0x3C,0x3D応答確認)						○						
160	7	1	Inter-byte-spaceのパラメータ振り(Inter-byte spaceは0以上)							○					
161		2	Response spaceのパラメータ振り((Response spaceは0以上)							○					
162		3	Inter-frame-spaceのパラメータ振り(Inter-frame spaceは0以上)							○					

図5　Excelで作成したマトリックス例
Excelを使用してマトリックスを作成すると，膨大な要求仕様やテスト項目の対応を簡単に整理することができる

- **テスト・パラメータの作成**── テスト・プログラムの基本的な動作（送受信）や動作オプション（通信速度，テストの期待値など）を文字や数値で定義したデータ．テスト項目ごとに作成する

こうすることで，一度テスト・プログラムを作成してしまえば，各テスト項目のテスト・パラメータを作成するだけでさまざまなテストに対応できます（**図7**）．また，テスト内容を変更する場合はテスト・パラメータの修正や追加だけを行えばよく，テスト・プログラムには影響を与えません．

このような方法は，単にテストの効率を向上させる以上の重要な意味を持っています．テスト・プログラムもプログラムである以上，不ぐあいを起こす可能性を持っています．みなさんの中にも，「対象プログラムとテスト・プログラムのどちらをテストしているのかわからない」という状況に陥った経験を持つ方がいることと思います．プログラムを1本に共通化することで，こうしたリスクを小さくすることができます．また，同じプログラムを繰り返し使用することで，テスト・プログラムの信頼性を向上させることもできます．

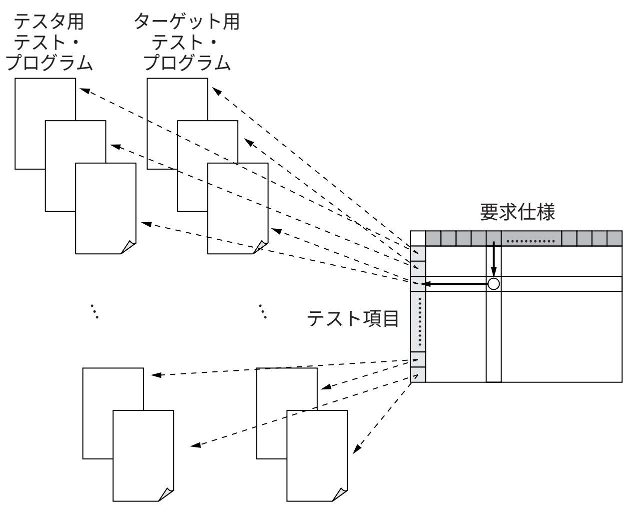


図6　テスト項目とテスト・パターン
テスト項目ごとにテスト・プログラムを作成していては，時間と手間がかかる．テスト項目の修正が発生すると，さらに工数がかさむ

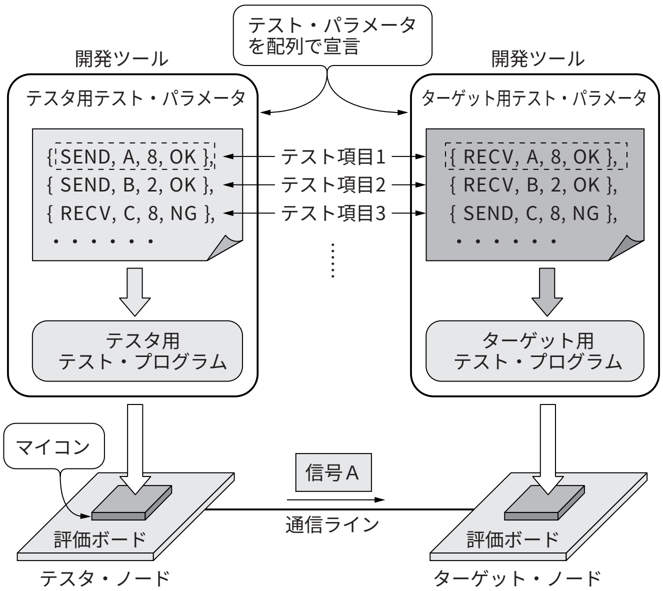


図8　テストの自動化
テスト・パラメータをテーブル化することで，テストの自動実行が可能になる

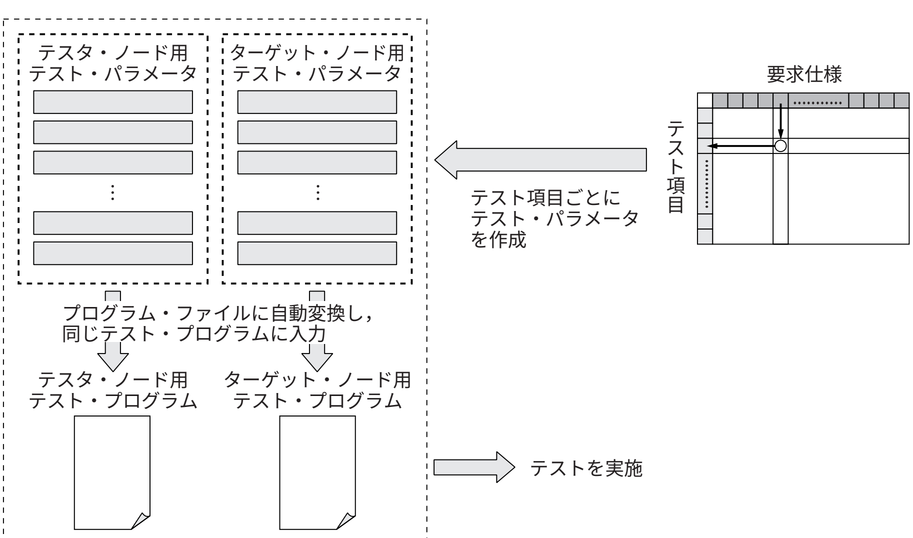


図7
テスト項目とテスト・パラメータ
すべてのテスト項目で共通のテスト・プログラムを使用すれば，あとはテスト項目ごとにその動作を決定するテスト・パラメータを作成するだけ．これにより，テスト項目の修正が発生してもテスト・パラメータを調整するだけで済む

● テスト作業を自動化する

テスト項目ごとにテスト・プログラムを作成する必要はなくなりましたが，各テスト作業を一つ一つ人手で実行していては膨大な時間と手間がかかってしまいます．この問題は，テスト・パラメータをテーブル化することで容易に解決でき，テストの連続実行が可能となります（**図8**）．ただし連続実行を行う場合，テスト項目ごとにテスタ・ノードとターゲット・ノードの間でテスト・プログラムの同期をとる必要があります．そのため，テスト項目の間で一定のウェイト処理や同期処理を行うか，通信信号とは別に同期信号を用意しなければなりません．

実際にテスト項目からテスト・パラメータを作成する例を以下に示します．例えば，テスト項目が「テスタ・ノードが通信速度9600bpsで送信した8ビット信号を，ターゲット・ノードが正常に受信できることを確認する」という場合を考えてみましょう．

まず，テスト・パラメータを作成します．テスト項目の通信フローは**図9**のようになります．ここからテスト・パラメータとして必要な動作やオプションを抽出し，Excelで作成したテスト・パラメータの表に記述します．

次に，テスト・プログラムで読み込めるように，表に記述したテスト・パラメータをプログラム・ファイルへ変換します．この変換はExcelの機能であるVisual Basicのマクロを利用して次のように行います（**図9**）．

1）プログラムの記述形式に変換――作成したテスト・パラメータをプログラムの記述形式に変換し，Excelの別シートに出力する．**図9**では，テスト・パラメータをC言語の構造体配列に変換している

2）ファイルへ出力――**1）**で変換したシートをVisual Basicのマクロを利用してファイルに出力する．テスタ用のプログラム・ファイルを**リスト1**に，ファイルを出力するマクロを**リスト2**に示す

● 結果をツールで照合して初めて「自動化」となる

テスト・プログラムは，テスト項目ごとにテスト結果をチェックします．テスト結果の期待値は，**図9**のようにテスト・パラメータの一つとしてテスト・プログラムに与えています．テスト・プログラムはテスト項目ごとに結果と期待値を比較し，次のように動作します（**図10**）．

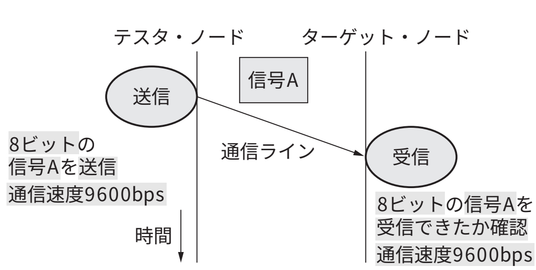
- 結果が真（OK）のとき――続けて次のテストを実行．すべてのテストが終了したときはテストの実行を停止する
- 結果が偽（NG）のとき――テストを停止し，失敗したテストの項目番号とエラー・コードを出力する

テスタ・ノードとターゲット・ノードのテスト・プログラムの両方がこの動作を行います．そのため，どちらかのテスト・プログラムが終了したときは，

- すべてのテストが正常に終了した
- テストに問題が発生した（結果が期待値と異なっている）

の二つの状態が考えられます．

このように，連続実行するテスト・プログラムを使って



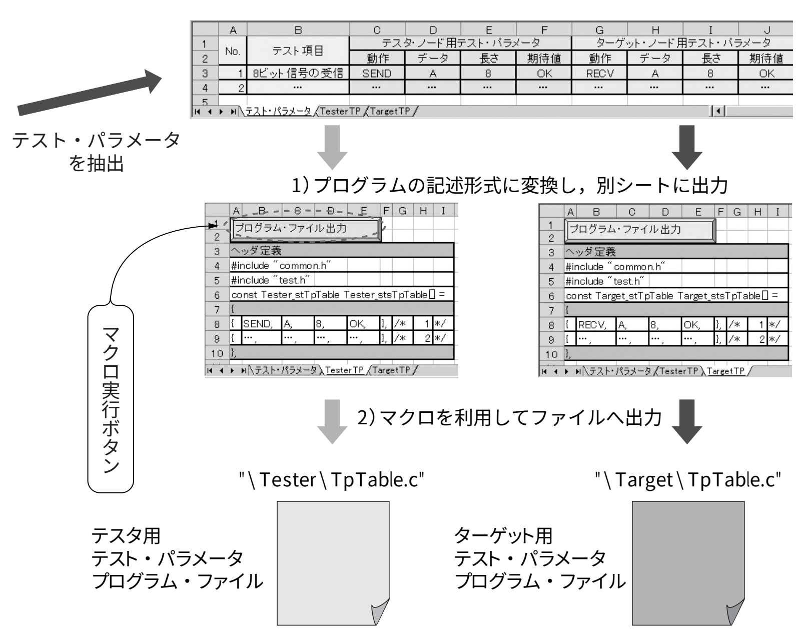


図9
テスト・パラメータの作成とプログラム記述形式への変換

テスト・パラメータを作成する前に通信フローを書いておくと，テストの内容を理解しやすい．後日テスト内容を再確認する場合にも役立つ．テスト・パラメータを直接プログラムで作成せず，Excelで記述する．これにより，テスタ・ノードとターゲット・ノードのテスト・パラメータの対応関係が見やすくなる．プログラムへの変換には，Excelが持っているVisual Basicマクロを利用する

結果のチェックも行うことで，テストの自動化が可能となります．

2 LINドライバのテストへ応用する

ここまで説明してきたテスト・システムを，LIN 2.0の通信ドライバに適用した例を紹介します．LINは車載ネットワークで利用する通信プロトコルの一つです．

通信を管理する一つのマスタ・ノードと複数のスレーブ・ノードがLINバスに接続され，各ノード間で通信を行います(**図11**)．

ここでテストのターゲットとなるのは，LIN 2.0スレーブ・ノードのドライバ・ソフトウェアです．スレーブ・ノードはマスタ・ノードから通信リクエスト(これをヘッダと言う)を受信し，リクエストに応じたデータ(これをレスポンスと言う)を送受信します．そこで今回は，スレーブ・ノードをテストするためのマスタ・ノードのドライバ・ソフトウェア(LIN 2.0マスタ・テスト治具ドライバ)を設計し，これをテスタとしました．

● マスタとスレーブのテスト・プログラムは別々に作成

テスト・システムの環境を**図12**に示します．ターゲット・ノードにはNECエレクトロニクスの8ビット・マイコ

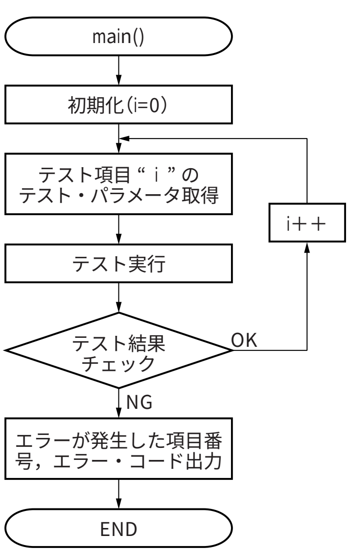


図10　テスト・プログラムの処理フロー

テスト・プログラムは，テスト・パラメータの読み込み，テストの実行，テスト結果のチェックを繰り返し行う

リスト1　テスタ・ノード用プログラム・ファイル

```
#include "common.h"
#include "test.h"
const Tester_stTpTable Tester_stsTpTable[] =
{
{SEND, A, 8, OK, }, /* 1 */
{•••, •••, •••, •••, }, /* 2 */
};
```

リスト2　プログラム・ファイル出力マクロ

```
Sub outputSlaveTP()

Dim buff As String
    Dim startCol  As Integer    ← 出力開始列
    Dim endCol    As Integer    ← 出力終了列
    Dim startRow  As Long       ← 出力開始行
    Dim endRow    As Long       ← 出力終了行
    Dim i         As Long
    Dim j         As Long
    Dim OutPutTP  As Variant

    OutPutTP = ActiveWorkbook.Path &
"\Tester\TpTable.c"    ← 出力ファイル名
    On Error GoTo Macro_Err
    Open OutPutTP For Output As #1
    On Error GoTo 0

    startCol = 1    ┐
    endCol   = 9    │ ← 出力開始列 出力終了列
    startRow = 4    │   出力開始行 出力終了行
    endRow   = 10   ┘

    For i = startRow To endRow
        buff = Cells(i, startCol).Value
        For j = startCol + 1 To endCol
            buff = buff & Cells(i, j).Value
        Next
        Print #1, buff    ← ファイルへ出力
    Next
    Close #1
    MsgBox “ファイル出力終了”
    Exit Sub

Macro_Err:

    MsgBox "OutPut File Error!"
    Exit Sub

End Sub
```

(On Error GoTo 0 から Exit Sub まで：通信パラメータ出力処理)

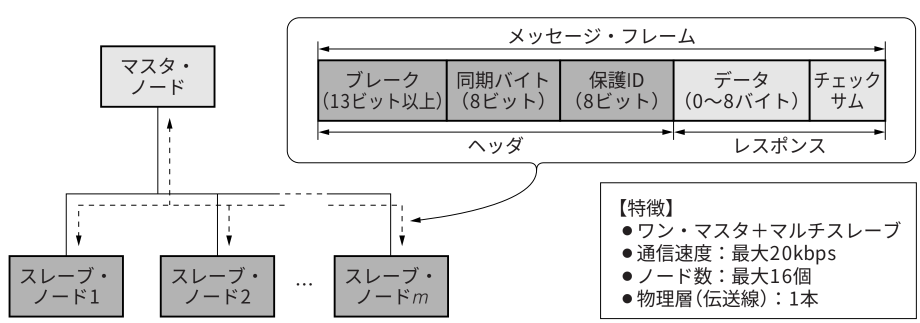


図11　LINの接続関係

LINは一つのマスタ・ノードと複数のスレーブ・ノードで構成される

4

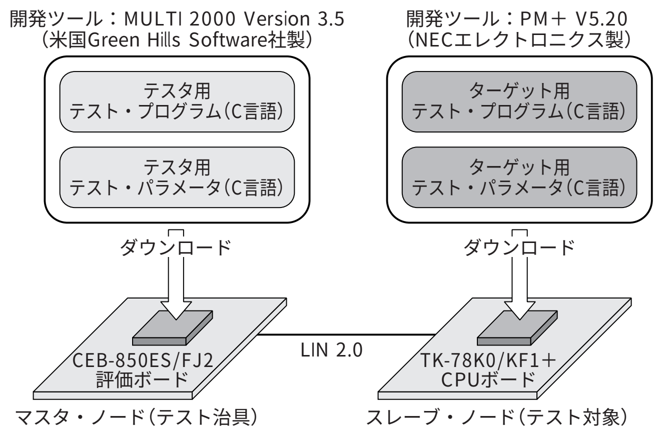
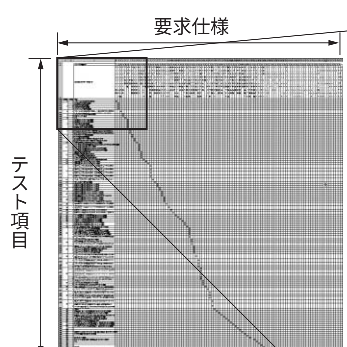


図12　テスト・システム環境
二つのマイコン・ボードをLINバスで接続し，テストを行う．テストの実行はマイコンのオンチップ・デバッグ機能を利用している

ン「μPD78F0148H（78K0/KF1+）」を，テスタ・ノードには同社の32ビット・マイコン「μPD70F3239（V850ES/FJ2）」を使用し，これらをLIN 2.0で接続しました．ターゲットにはテスト対象となるドライバ・ソフトウェアとテスト・プログラムを同社の「統合開発環境プラットフォームPM＋（プロジェクト・マネージャ）」を用いてダウンロードします．一方，テスタ・ノードにはテスト治具用ドライバとテスト・プログラムを米国Green Hills Software社のソフトウェア開発環境「MULTI 2000」を用いてダウンロードし，テストを実施します．テストはターゲット，テスタともにツールのオンチップ・デバッグ機能を用いて行います．この環境を用いたLINドライバのテストの流れは以下のとおりです．

1) **テスト項目の抽出とテスト仕様書の作成**―― Excelで作成したマトリックスを用いて，LIN 2.0スレーブ・ドライバの仕様書からテスト項目を抽出する（**図13**）
2) **テスト・パラメータ作成とプログラム・ファイル出力**―― 各テスト項目を実現するため，マスタ・ノードとスレーブ・ノードの動作をテスト・パターンのフォーマットで記述していく（**図14**）．これは，抽出したテスト項目とは別のExcelシートに記述する．記述後，マクロでプログラム・ファイルを出力する（**リスト2**を参照）
3) **テスト・プログラムの作成**―― テスト・パラメータのプログラム・ファイルを読み取り，テスト・プログラムを作成する．このプログラムは，マスタ・ノードとスレーブ・ノードで別々に作成する．マスタ・ノードのテスト・プログラムを**リスト3**（pp.72-73）に示す．テスト・プログラムはテスト・パラメータの構造体配列から数値やマクロ値を取得し，通信を行う．通信結果が期待値と異なる場合，エラー発生関数をコールし，テストを停止する
4) **テストの実施**―― 作成したテスト・パラメータのプログラム・ファイルとテスト・プログラムを，開発ツールを用いてテスタ・ノードとターゲット・ノードのマイコンにダウンロードし，テストを実行する．テスト・パターン作成直後はちょっとした記述ミスや通信フローの誤認で動かないことが多いため，ツールのステップ実行機能を用いてテスト項目ごとに通信を確認する．必



		スレーブ機能番号	1	2	3	4	5	6	7	8	9	10
		仕様書該当箇所＼機能名称	Singnal定義	Singnalとバイトアレイ	Singnalの一貫性	Singnal境界	フレーム定義	レスポンス機能	スペース管理	UART送受信機能	Break受信機能	Synch受信機能
大項目	中項目	内容										
1	1	1フレーム中複数信号の送受信	○									
2	1	boot型信号の送受信確認		○								
	2	8bit信号の送受信確認		○								
	3	16bit信号の送受信確認		○								
	4	バイトアレイの送受信確認		○								
3	1	1フレーム中に複数信号が存在し、一部の信号だけが更新された場合でも送受信はフレーム単位で実施されることを確認			○							
4	1	1バイトまたぎ信号の送受信確認				○						
	2	LSBファースト送信				○						
5	1	なし					○					

対応した項目を○で表示

図13
LIN 2.0スレーブ・ドライバのテスト項目の抽出
テスト項目は膨大な数（約1,500）になった

要であればオシロスコープなどを用いてLINバスの状態も確認すること．ステップ実行でテストの正常動作が確認できたら，テストを連続実行する

● テスト項目からテスト・パターンを作成する

次に，テスト項目からテスト・パターンを作成する例を説明します．

ここでは，「ブール型信号の正常受信を確認する」というテスト項目を取り上げます．LIN 2.0で定義されているブール型信号（1ビット幅の信号）をマスタ・ノードから送信し，スレーブ・ノードのアプリケーションがLIN 2.0で定めるAPI（application programming interface）関数を用いて，信号の正常受信を行えるかどうかを確認するテストを行います．

このテストの通信フローを**図15**に示します．マスタ側のテスト・プログラムは，ブール型信号の受信をリクエストするヘッダをスレーブ・ノードに送信し，それに続いてブール型信号を含むレスポンスを送信します．スレーブ・ノードのテスト・プログラムはLIN 2.0で定められている`l_flg_tst`関数をコールしてブール型信号の更新をチェックします．そして信号更新時に`l_bool_rd`関数をコールしてブール型信号を読み出します．

図15の通信フローに従って作成したテスト・パターンを**図16**に示します．マスタ・ノードのテスト・パターンにはブール型信号を送信するSENDを動作に記述し，送信信号値を‘1’と指定しています．スレーブ・ノードには関数として信号を受信するためのAPI関数（`l_flg_tst`と`l_bool_rd`）を記述し，各関数の戻り値を期待値としています．

● どうやって通信エラーを発生させるか

テストでは異常通信をわざと発生させて，そのときの動



スレーブ					
関数	信号 or ステータス	LENG or 同期待ち	データ(16進数),未使用はDUMMY	ループ回数 or その他のパラメータ	ループ内で実行する関数
l_flg_tst	DUMMY	DUMMY	DUMMY	0	l_bool_rd
l_flg_tst	M0A	WAITRD	DUMMY	DEFAULT	DEFAULT
l_bool_rd	M0A	1	0x01	DEFAULT	DEFAULT
l_flg_tst	M0B	WAITRD	DUMMY	DEFAULT	DEFAULT
l_bool_rd	M0B	1	0x01	DEFAULT	DEFAULT
l_flg_tst	M0C	WAITRD	DUMMY	DEFAULT	DEFAULT
l_bool_rd	M0C	1	0x01	DEFAULT	DEFAULT
l_flg_tst	M0A	WAITRD	DUMMY	DEFAULT	DEFAULT

大項目	中項目	内容	小項目	内容	動作	PID	データ	SUM	通信結果の期	バイト数	ループ 開始値
				パラメータデフォルト値	SEND	0x80	0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00	OK	OK	8	0
	1	boot型信号の送受信確認(データ依存も確認)	1	マスターからスレーブへbool型信号'1'を含む8バイトデータを送信(最下位,中間,最上位)	SEND	0x80	0x01,0x22,0x33,0x80,0x55,0x66,0x77,0x80	OK	OK	8	DUMMY
			2	マスターからスレーブへbool型信号'0'を含む8バイトデータを送信(最下位,中間,最上位)	SEND	0x80	0xFE,0x22,0x33,0x7E,0x55,0x66,0x77,0x7F	OK	OK	8	DUMMY
			3	スレーブからマスターへbool型信号'1'を含む8バイトデータを送信(最下位ビット)	RECV	0xC1	0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00	OK	OK	8	DUMMY
			4	スレーブからマスターへbool型信号'0'を含む8バイトデータを送信(最下位ビット)	RECV	0xC1	0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00	OK	OK	8	DUMMY

図14
LIN 2.0スレーブ・ドライバのテスト・パラメータの作成
テスト項目のテスト・パラメータを作成する．テスト項目が膨大であってもテスト・パラメータ方式であれば実現しやすい（マスタ・ノード用テスト・パラメータは25個，スレーブ・ノード用テスト・パラメータは6個）

作を確認することがあります．LIN 2.0 ドライバのテストでもチェックサム・エラーやフレーミング・エラー，ビット・エラーなどの異常通信を再現し，確認しています．例としてビット・エラーの再現方法を示します．

LIN 2.0 のビット・エラーとは，自分が送信したデータと，送信データをループバックして受信したデータに不一致が発生することです(**図17**)．通常，複数のノードが同時に異なったデータを送信することで発生します．実施したテストでは，最初にレスポンス送信をリクエストするヘッダをスレーブ・ノードに送信します．そして，ヘッダに

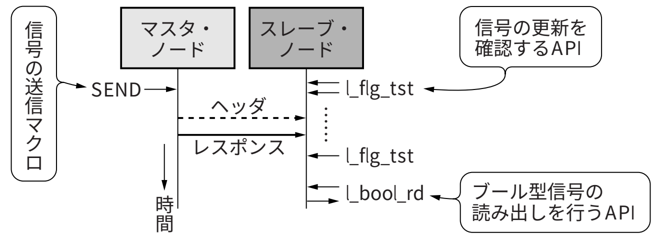


図15　テストの通信フローの作成
通信フローを作成することで，複雑なテストの流れを容易に表現できる

リスト3　マスタ・ノードのテスト・プログラム

```
#include "TpTable.h"        /* テスト・テーブル情報             */

#define STARTNO     1       /* 開始テスト番号 (0x0001～0xffff) */
#define ENDNO       0xffff  /* 終了テスト番号 (0x0001～0xffff) */

void main()
{
    ApLst_vosInitTest();    /* テスト・プログラム初期化 */

    /* テスト開始 */
    for( TableNo = (STARTNO - 1); TableNo < ENDNO; TableNo++)
    {
        /* テスト内容の判別とテストの実行   */
        ApLst_vosTest( (Tester_stTpTable *)&Tester_stsTpTable[TableNo],
                       (DrvLt_stLinprm *)&(Tester_stsTpTable[TableNo].ApLst_stLinprm) );

        DrvLt_vosCtrlWait( 5000000L ); /* テスト項目ごとに1[ms]のウェイト */
    }

    return;
}

/* エラー関数   */
static void ApLst_vosSetFail(void)
{
    while(1);           /* テスト・エラー・ブレークを設定 */
}

/* テスト・プログラム本体   */
static void ApLst_vosTest(Tester_stTpTable *sttLtTable, DrvLt_stLinprm *psttLinprm )
{
    u2 i;
    u1 u1tSum;          /* 送信チェックサム・データ       */
    u1 u1tRecvData[8];  /* 受信データ格納バッファ         */
    u1 u1tRecvSum;      /* 受信チェックサム格納バッファ   */
    s1 s1tResult;       /* 実行結果                       */

    /* チェックサム計算 */
    u1tSum = ApLst_u1sCalcSum(&(sttLtTable->u1Data[0]), sttLtTable->u1Length, sttLtTable->u1Pid);
    u1tSum = ~u1tSum;

    /* テスト・パラメータの“チェックサム期待値”がERRORのときはエラーとなるチェックサムを生成  */
    if( sttLtTable->s1SumExpect == ERROR )
    {
        u1tSum++;        /* チェックサム結果ERROR設定 */
    }

    /* テスト・パラメータの“動作”を判別して対応したテストを実行 */
    switch(sttLtTable->u1Dir)
    {
        /* ヘッダとレスポンスの送信テスト */
        case SEND:
            s1tResult = DrvLt_s1gSendMessage(sttLtTable->u1Pid,
                                             sttLtTable->u1Data,
                                             u1tSum,
                                             sttLtTable->u1Length,
                                             psttLinprm );
```

リクエスト内容を決定する識別子

送信データの期待値（正常に送信できるかどうか）

スレーブ用テスト・プログラムが実行するLIN 2.0のAPI関数名

構造体配列の要素番号

テスト項目	マスタ・ノード						スレーブ・ノード				
	動作	PID	データ	SUM	期待値	No	関数	信号	長さ/フラグ	期待値	No
ブール型信号の受信確認	SEND	0x80	0x01，0x00，…，0x00	OK	OK	1	l_flg_tst	M0A	1	1	1
							l_bool_rd	M0A	1	1	2

信号送信のマクロ

ブール型信号を含むレスポンスのデータ・パターン

レスポンスに付加するチェックサム・エラーの有無

APIの実行対象の信号名

読み出す信号更新フラグの期待値（一致するまでループ）

読み出す信号のデータ幅

読み出す信号の期待値

図16　テスト・パターンの作成

通信フローに従ってテスト・パターンを作成する．一つの通信でテストが完了しないときは複数行に定義する

```
            /* テスト結果がテスト・パラメータの “ 期待値 ” と異なる場合はエラー関数をコール */
            if( s1tResult != sttLtTable->s1ComExpect )
            {
                ApLst_vosSetFail();
            }

            break;

        /* ヘッダの送信とレスポンスの受信テスト  */
        case RECV:
            s1tResult = DrvLt_s1gRecvMessage(sttLtTable->u1Pid,
                                             u1tRecvData,
                                             &u1tRecvSum,
                                             sttLtTable->u1Length,
                                             psttLinprm,
                                             &ApLst_stsRecTime );

            /* テスト結果がテスト・パラメータの “ 期待値 ” と異なる場合はエラー関数をコール */
            if( s1tResult != sttLtTable->s1ComExpect )
            {
                ApLst_vosSetFail();
            }

            /* “ 期待値 ” が正常終了の場合，テストの実行結果が正常かどうかをチェック */
            if( s1tResult == OK )
            {
                /* 受信データ・チェック  */
                for( i = 0; i < sttLtTable->u1Length; i++ )
                {
                    if(u1tRecvData[i] != sttLtTable->u1Data[i])
                    {
                        ApLst_vosSetFail();
                    }
                }

                /* チェックサム・エラー・チェック  */
                if( u1tSum != u1tRecvSum )
                {
                    ApLst_vosSetFail();
                }

                break;
            }

        … 中略 …

        /* 該当するテストがない場合エラー関数をコール */
        default:

            ApLst_vosSetFail();

            break;
    }

    return;
}
```

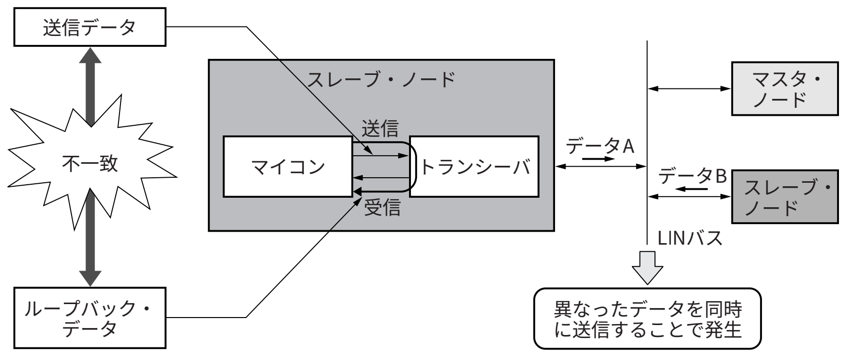


図17　ビット・エラー
LINに限らず，異常系の扱いはテストのポイントの一つ

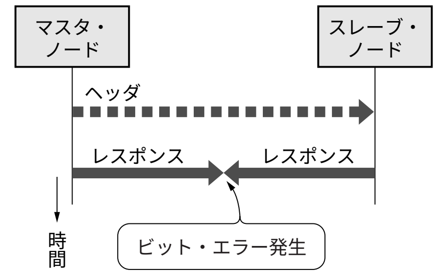


図18　ビット・エラーの再現
同一タイミングで送信を行わせることで，衝突を実現させた

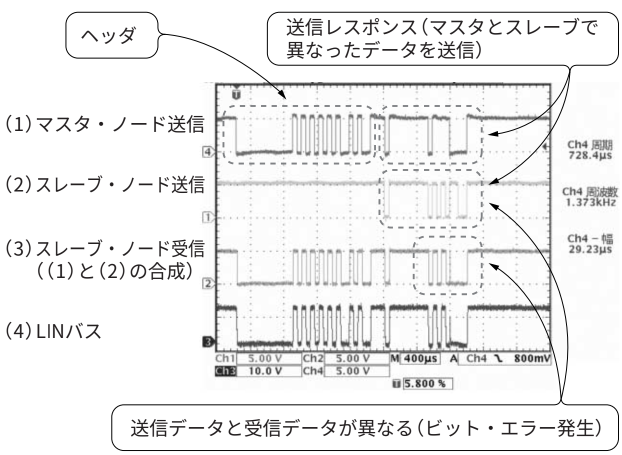


図19　ビット・エラー波形
エラー発生時，実際に何が起こっているのかを確認することは重要である

続いてわざとマスタからもレスポンスを送信することで，ビット・エラーを実現しています（**図18**，**図19**）．

●「テストも設計の一つ」と心得るべし

本稿で紹介したLINドライバでは，約1,500項目のテストを実施しています．また，LINのような汎用ドライバ・ソフトウェアは，マイコンの品種ファミリへの移植展開が求められます．さらには提供先ごとのカスタマイズによる改造が伴うこともあり，これらすべてに対してすべてのテストを人間の操作作業のみで実施しようとすると，何年もかかってしまいます．また，テストもれによる不ぐあいなどが発生した場合，展開した品種のすべてを再検査することとなり，その影響は計りしれません．

今回の自動テスト・システムの開発には，およそ2人月の工数を投入していますが，その結果，1品種当たりのテストは数日以内で実行可能となりました．

仕様からの項目抽出作業やテスト・システムの構築は，多くの場合，かなりの工数を必要とし，一見開発コストの悪化要因に見えます．しかし，品種展開のない単品ソフトウェアの場合であっても，不ぐあいや仕様変更によるリワークは日常茶飯事です．リワーク回数とテスト工数の積を見積もった場合，多くの開発において今回のような施策は投入工数を上回る効果をもたらすと思います．

テストは，開発スケジュールの最後に余った時間に力業で行うのではなく，プロジェクト当初から十分計画的に考えておくことがたいせつです．テストは実際のプログラム作成と並んで，別視点での「もう一つの設計」と考えるべきでしょう．

参考・引用*文献

(1) * LIN Consortium ; LIN Specification Package Revision 2.0, Sep. 23, 2003.

なの・ひびき
NECマイクロシステム（株）

<筆者プロフィール>
名野響．デバイス設計会社に入社したものの，見えない力に引きずられて組み込み系ソフトウェア・エンジニアに．